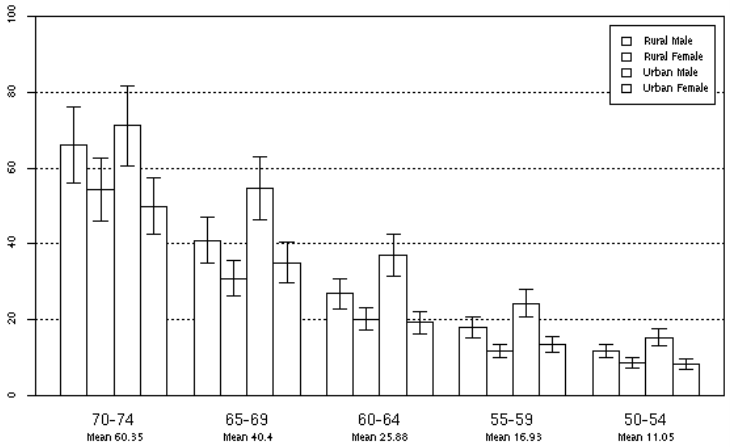
REVOLUTION ANALYTICS



Scalable Data Analysis in R

Lee E. Edlefsen
Chief Scientist
UserR! 2011

Introduction

- Our ability to collect and store data has rapidly been outpacing our ability to analyze it
- We need scalable data analysis software
- R is the ideal platform for such software: universal data language, easy to add new functionality, powerful, flexible, forgiving
- I will discuss the approach to scalability we have taken at Revolution Analytics with our package **RevoScaleR**

RevoScaleR package

- Part of Revolution R Enterprise
- Provides data management and data analysis functionality
- Scales from small to huge data
- Is internally threaded to use multiple cores
- Distributes computations across multiple computers (in version 5.0 beta on Windows)
- Revolution R Enterprise is free for academic use

Overview of my talk

- What do I mean by scalability?
- Why bother?
- Revolution's approach to scalability:
  - R code stays the same as much as possible
  - "Chunking" – operate on chunks of data
  - Parallel external memory algorithms
  - Implementation issues
- Benchmarks

Scalability

- From small in-memory data.frames to multi-terabyte data sets distributed across space and even time
- From single cores on a single computer to multiple cores on multiple computers/nodes
- From local hardware to remote clouds
- For both data management and data analysis

Keys to scalability

- Most importantly, must be able to process more data than can fit into memory on one computer at one time
- Process data in “chunks”
- Up to a point, the bigger the chunk the better
  - Allows faster sequential reads from disk
  - Minimizes thread overhead in memory
  - Takes maximum advantage of R’s vectorized code
- Requires “external memory algorithms”

Huge data sets are becoming common

- Digital data is not only cheaper to store, it is cheaper to collect
  - More data is collected digitally
  - Increasingly more data of all types is being put directly into data bases
- It is easier now to merge and clean data, and there is a big and growing market for such data bases

Huge benefits to huge data

- More information, more to be learned
- Variables and relationships can be visualized and analyzed in much greater detail
- Can allow the data to speak for itself; can relax or eliminate assumptions
- Can get better predictions and better understandings of effects

Two huge problems: capacity and speed

- **Capacity**: problems handling the size of data sets or models
  - Data too big to fit into memory
  - Even if it can fit, there are limits on what can be done
  - Even simple data management can be extremely challenging
- **Speed**: even without a capacity limit, computation may be too slow to be useful

We are currently incapable of analyzing a lot of the data we have

- The most commonly-used statistical software tools either fail completely or are too slow to be useful on huge data sets
- In many ways we are back where we were in the ‘70s and ‘80’s in terms of ability to handle common data sizes
- Fortunately, this is changing

Requires software solutions

- For decades the rising tide of technology has allowed the same data analysis code to run faster and on bigger data sets
- That happy era has ended
- The size of data sets is increasing much more rapidly than the speed of single cores, of I/O, and of RAM
- Need software that can use multiple cores, multiple hard drives, and multiple computers

High Performance Analytics: HPA

- HPA is HPC + Data
- High Performance Computing is CPU centric
  - Lots of processing on small amounts of data
  - Focus is on cores
- High Performance Analytics is data centric
  - Less processing per amount of data
  - Focus is on feeding data to the cores
    - On disk I/O
    - On efficient threading, data management in RAM

Revolution's approach to HPA and scalability in the RevoScaleR package

- User interface
  - Set of R functions (but mostly implemented in C++)
  - Keep things as familiar as possible
- Capacity and speed
  - Handle data in large "chunks" – blocks of rows and columns
  - Use parallel external memory algorithms

Some interface design goals

- Should be able to run the same analysis code on a small data.frame in memory and on a huge distributed data file on a remote cluster
- Should be able to do data transformations using standard R language on standard R objects
- Should be able to use the R formula language for analysis

Sample code for logit on laptop

```
## Standard formula language
## Allow transformations in the formula or
#   in a function or a list of expressions
## Row selections also allowed (not shown)


rxLogit(ArrDelay>15 ~ Origin + Year +
   Month + DayOfWeek + UniqueCarrier +
   F(CRSDepTime), data=airData)
```

Sample code for logit on a cluster

```
# Just change the "compute context"
rxOptions(computeContext = myClust)

# Otherwise, the code is the same
rxLogit(ArrDelay>15 ~ Origin + Year +
   Month + DayOfWeek + UniqueCarrier +
   F(CRSDepTime), data=airData)
```

Sample data transformation code

```
## Standard R code on standard R objects
## Function or a list of expressions
## In separate data step or in analysis call

 transforms <- list(
DepTime = ConvertToDecimalTime(DepTime),
Late = ArrDelay > 15
)
```

Chunking

- Operate blocks of rows for selected columns
- Huge data can be processed a chunk at a time in a fixed amount of RAM
- Bigger is better up to a point
  - Makes it possible to take maximum advantage of disk bandwidth
  - Minimizes threading overhead
  - Can take maximum advantage of R's vectorized code

The basis for a solution for capacity, speed, distributed and streaming data – PEMA's

- **Parallel external memory algorithms (PEMA's)** allow solution of both capacity and speed problems, and can deal with distributed and streaming data
- External memory algorithms are those that allow computations to be split into pieces so that not all data has to be in memory at one time
- It is possible to "automatically" parallelize and distribute such algorithms

External memory algorithms are widely available

- Some statistics packages originating in the 70's and 80's, such as SAS and Gauss, were based almost entirely on external memory algorithms
- Examples of external memory algorithms:
  - data management (transformations, new variables, subsets, sorts, merges, converting to factors)
  - descriptive statistics, cross tabulations
  - linear models, glm, mixed effects
  - prediction/scoring
  - many maximum likelihood and other optimization algorithms
  - many clustering procedures, tree models

Parallel algorithms are widely available

- Parallel algorithms are those that allow different portions of a computation to be done “at the same time” using different cores
- There has been an lot of research on parallel computing over the past 20-30 years, and many helpful tools are now available

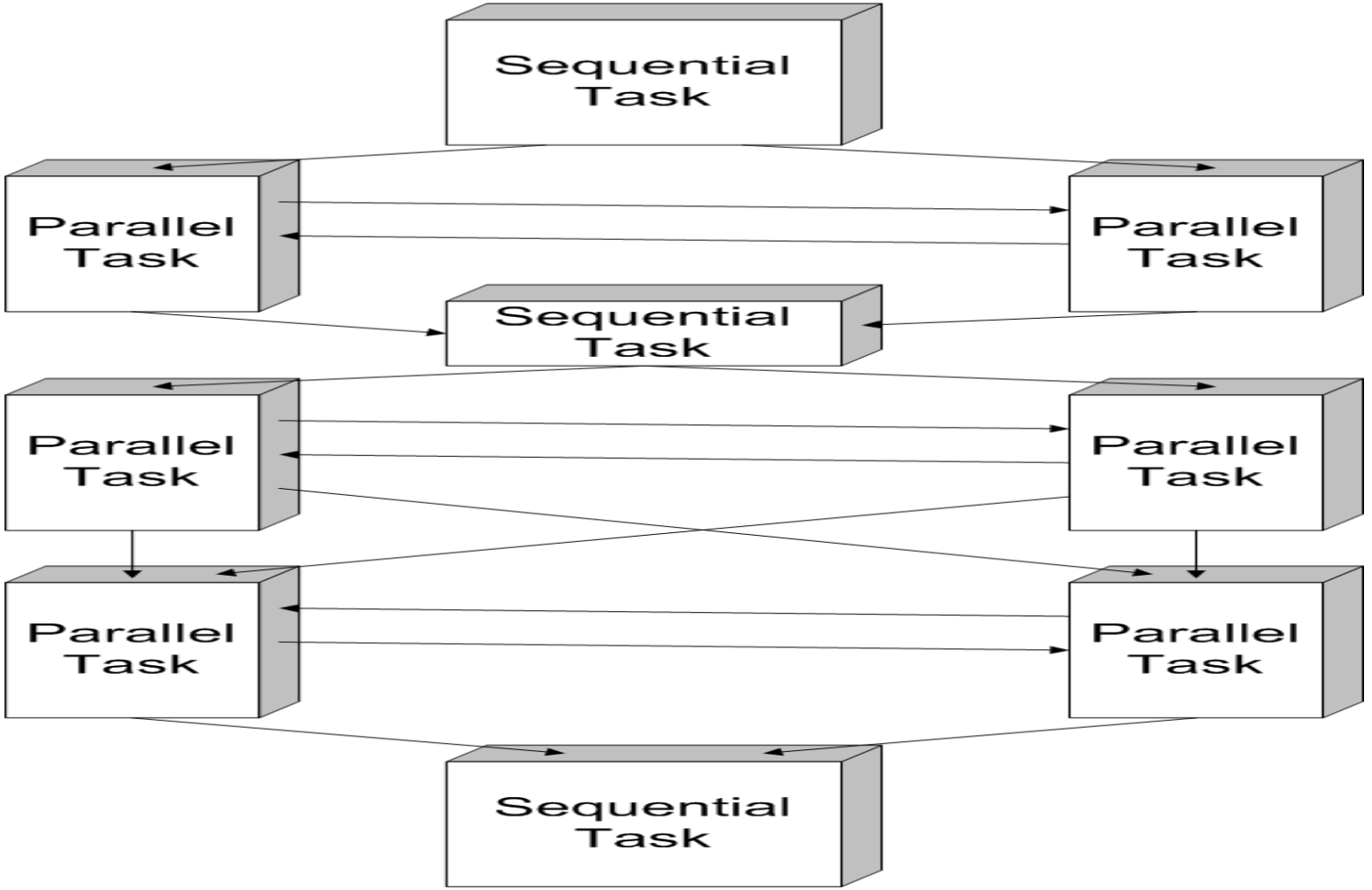
Not much literature on PEMA's

- Unfortunately, there is not much literature on Parallel External Memory Algorithms, especially for statistical computations
- In what follows, I outline an approach to PEMA's for doing statistical computations

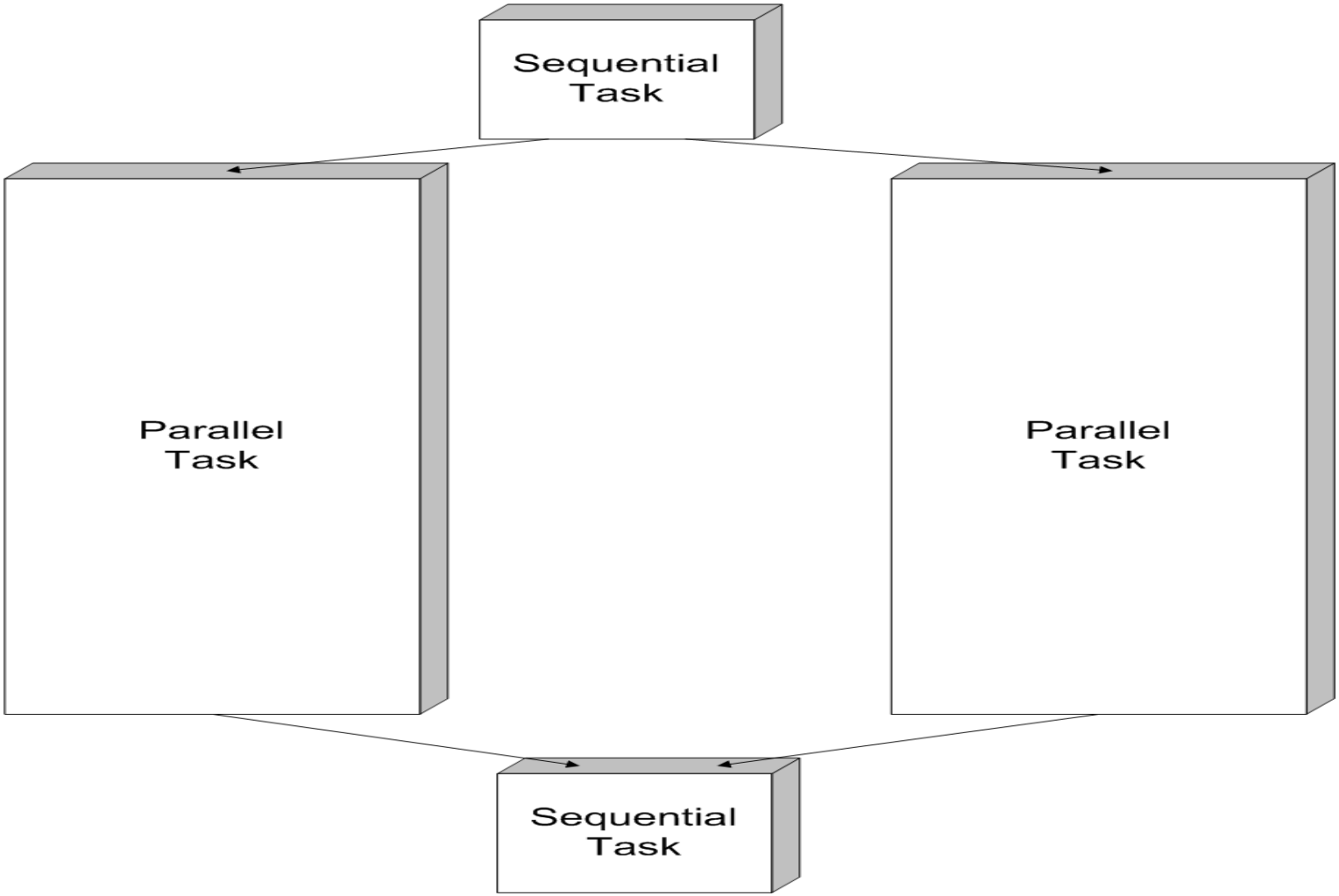
Parallelizing external memory algorithms

- Parallel algorithms **split a job into tasks** that can be run simultaneously
- External memory algorithms **split a job into tasks** that operate on separate blocks data
- External memory algorithms usually are run sequentially, but with proper care most can be **parallelized efficiently and "automatically"**
- The key is to **split the tasks appropriately** and to minimize inter-thread communication and synchronization

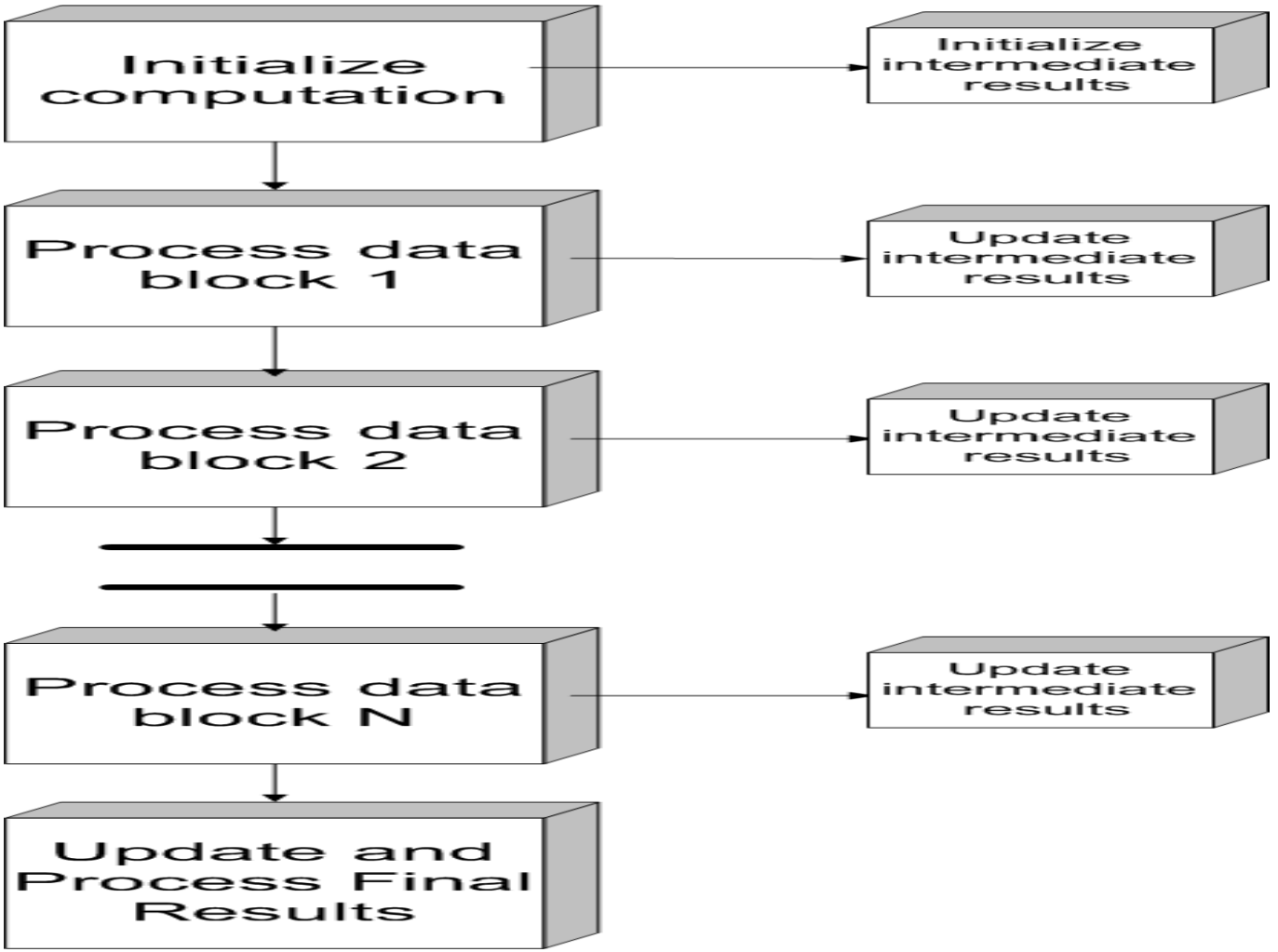
Parallel Computations with Synchronization and Communication



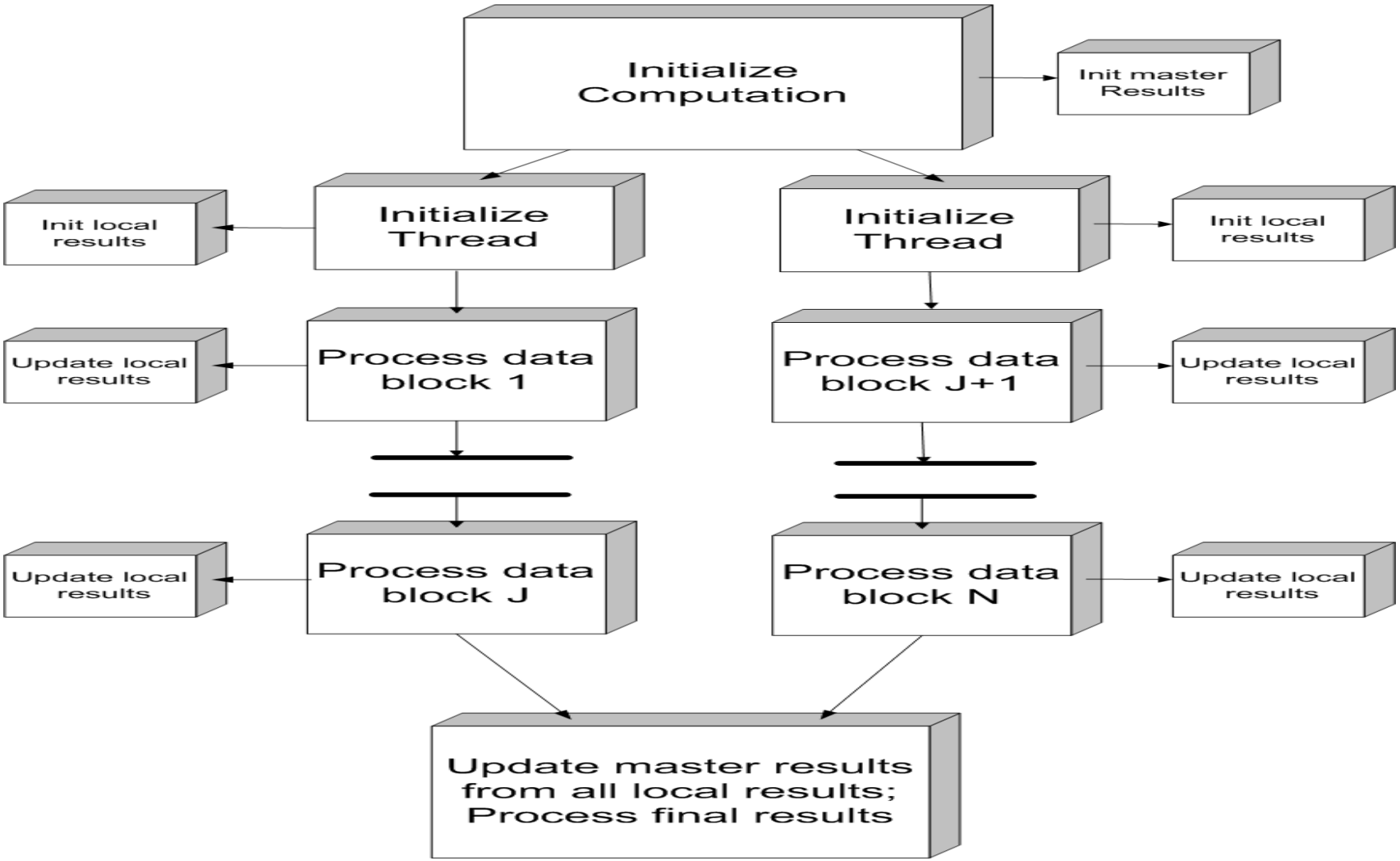
Embarrassingly Parallel Computations



Sequential External Memory Computations



Parallel External Memory Computations



Example external memory algorithm for the mean of a variable

- **Initialization** task: total=0, count=0
- **Process data** task: for each block of x; total = sum(x), count=length(x)
- **Update results** task: combined total = sum(all totals), combined count = sum(all counts)
- **Process results** task: mean = combined total / combined count

PEMA's in RevoScaleR

- Analytics algorithms are implemented in a framework that automatically and efficiently parallelizes code
- Key is arranging code in virtual functions:
  - Initialize: allows initializations
  - ProcessData: process a chunk of data at a time
  - UpdateResults: updates one set of results from another
  - ProcessResults: any final processing

PEMA's in RevoScaleR (2)

- Analytic code is completely independent of:
  - Data path code that feeds data to ProcessData
  - Threading code for using multiple cores
  - Inter-computer communication code for distributing across nodes
- Communication among machines is highly abstracted (currently can use MPI, RPC)
- Implemented in C++ now, but we plan to add an implementation in R

Storing and reading data

- ScaleR can process data from a variety of sources
- Has its own optimized format (XDF) that is especially suitable for chunking
- Allows rapid access to blocks of rows for selected columns
- The time it takes to read a block of rows is essentially independent of the total number of rows and columns in the file

The XDF file format

- Data is stored in blocks of rows per column
- “Header” information is stored at the end of the file
- Allows sequential reads; tens to hundreds of thousands of times faster than random reads
- Essentially unlimited in size
  - 64 bit row indexing per column
  - 64 bit column indexing (but the practical limit is much smaller)

The XDF file format (2)

- Allows wide range of storage formats (1 byte to 8 byte signed and unsigned integers; 4 and 8 byte floats; variable length strings)
- Both new rows and new columns can be added to a file without having to rewrite the file
- Changes to header information (variable names, descriptions, factor levels, and so on) are extremely fast

Overview of data path on a computer

- DataSource object reads a chunk of data into a DataSet object on I/O thread
- DataSet is given to transformation code (data copied to R for R transformations); variables and rows may be created, removed
- Transformed DataSet is virtually split across computational cores and passed to ProcessData methods on different threads
- Any disk output is stored until I/O thread can write it to output DataSource

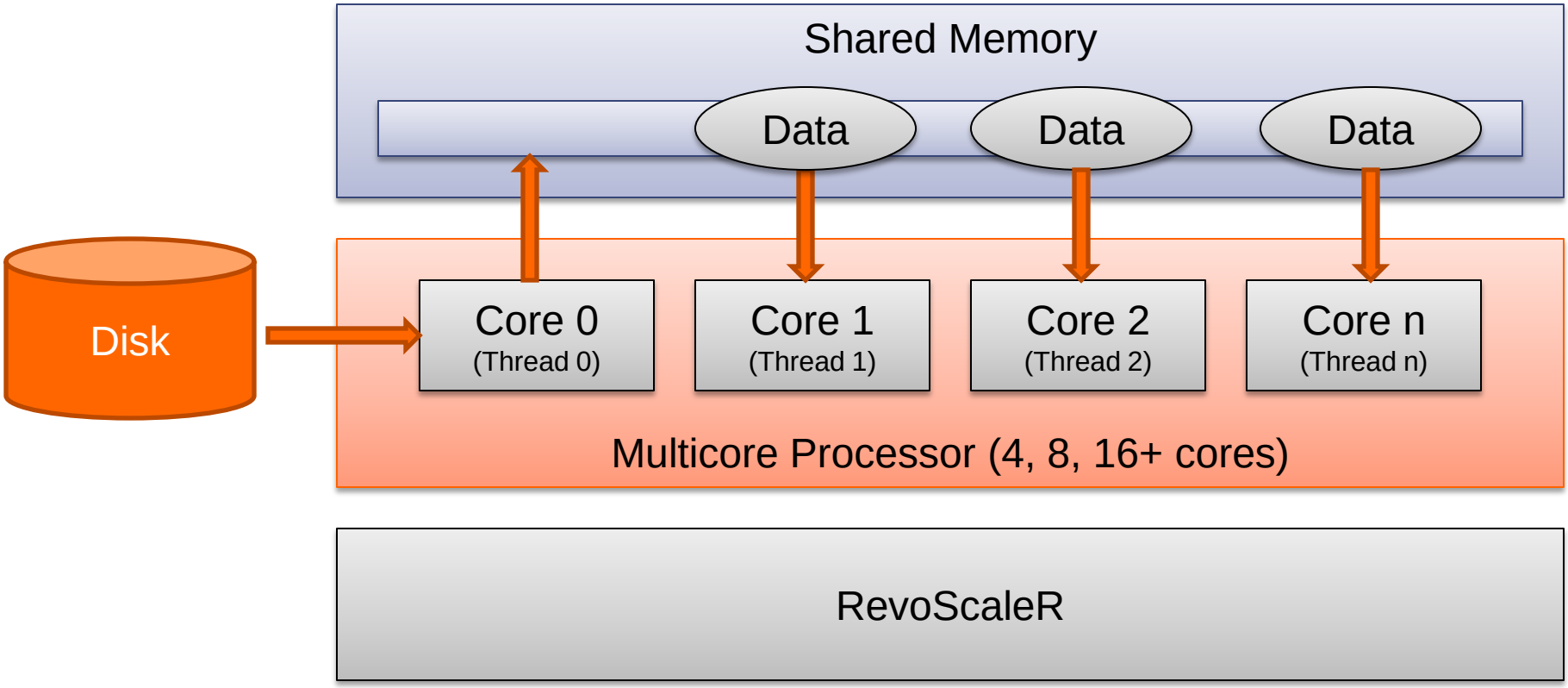
Handling data in memory

- Use of appropriate storage format reduces space and reduces time to move data in memory
- Data copying and conversion is minimized
- For instance, when adding a vector of unsigned shorts to a vector of doubles, the smaller type is not converted until loaded into the CPU

Use of multiple cores per computer

- Code is internally “threaded” so that inter-process communication and data transfer is not required
- One core (typically) handles I/O, while the other cores process data from the previous read
- Data is virtually split across computational cores; each core thinks it has its own private copy

RevoScaleR – Multi-Threaded Processing

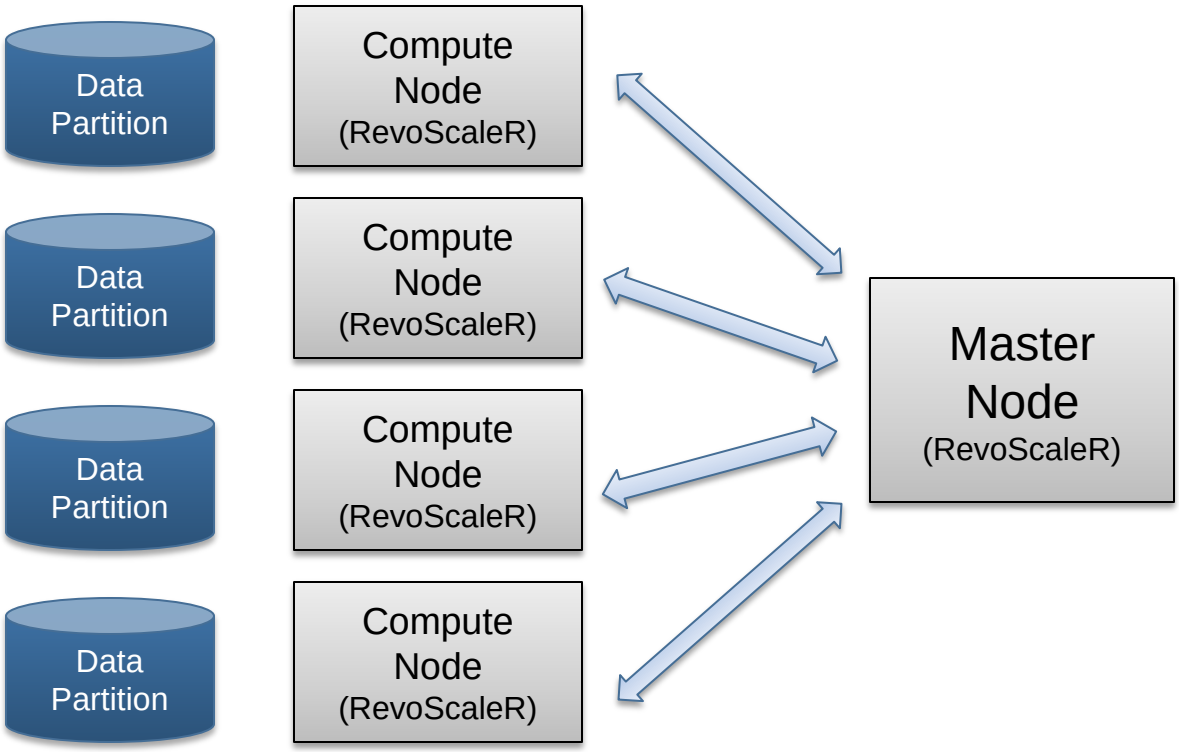


- A RevoScaleR algorithm is provided a data source as input
- The algorithm loops over data, reading a block at a time. Blocks of data are read by a separate worker thread (Thread 0).
- Other worker threads (Threads 1..n) process the data block from the previous iteration of the data loop and update intermediate results objects in memory
- When all of the data is processed a master results object is created from the intermediate results objects

Use of multiple computers

- Key to efficiency is minimizing data transfer and communication
- Locality of data!
- For PEMA's, the master node controls computations, telling workers where to get data and what computations to do
- Intermediate results on each node are aggregated across cores
- Master node gathers all results, checks for convergence, and repeats if necessary

RevoScaleR – Distributed Computing



- Portions of the data source are made available to each compute node
- RevoScaleR on the master node assigns a task to each compute node
- Each compute node independently processes its data, and returns it's intermediate results back to the master node
- master node aggregates all of the intermediate results from each compute node and produces the final result

Data management capabilities

- Import from external sources
- Transform and clean variables
- Code and recode factors
- Missing values
- Validation
- Sort (huge data, but not distributed)
- Merge
- Aggregate, summarize

Analysis algorithms

- Descriptive statistics (rxSummary)
- Tables and cubes (rxCube, rxCrossTabs)
- Correlations/covariances (rxCovCor, rxCor, rxCov, rxSSCP)
- Linear regressions (rxLinMod)
- Logistic regressions (rxLogit)
- K means clustering (rxKmeans)
- Predictions (scoring) (rxPredict)
- Other algorithms are being developed

Benchmarks using the airline data

- Airline on-time performance data produced by U.S. Department of Transportation; used in the ASA Data Expo 09
- 22 CSV files for the years 1987 – 2008
- 123.5 million observations, 29 variables, about 13 GB
- For benchmarks, sliced and replicated to get files with 1 million to about 1.25 billion rows
- 5 node commodity cluster (4 cores @ 3.2GHz & 16 GB RAM per node); Windows HPC Server 2008

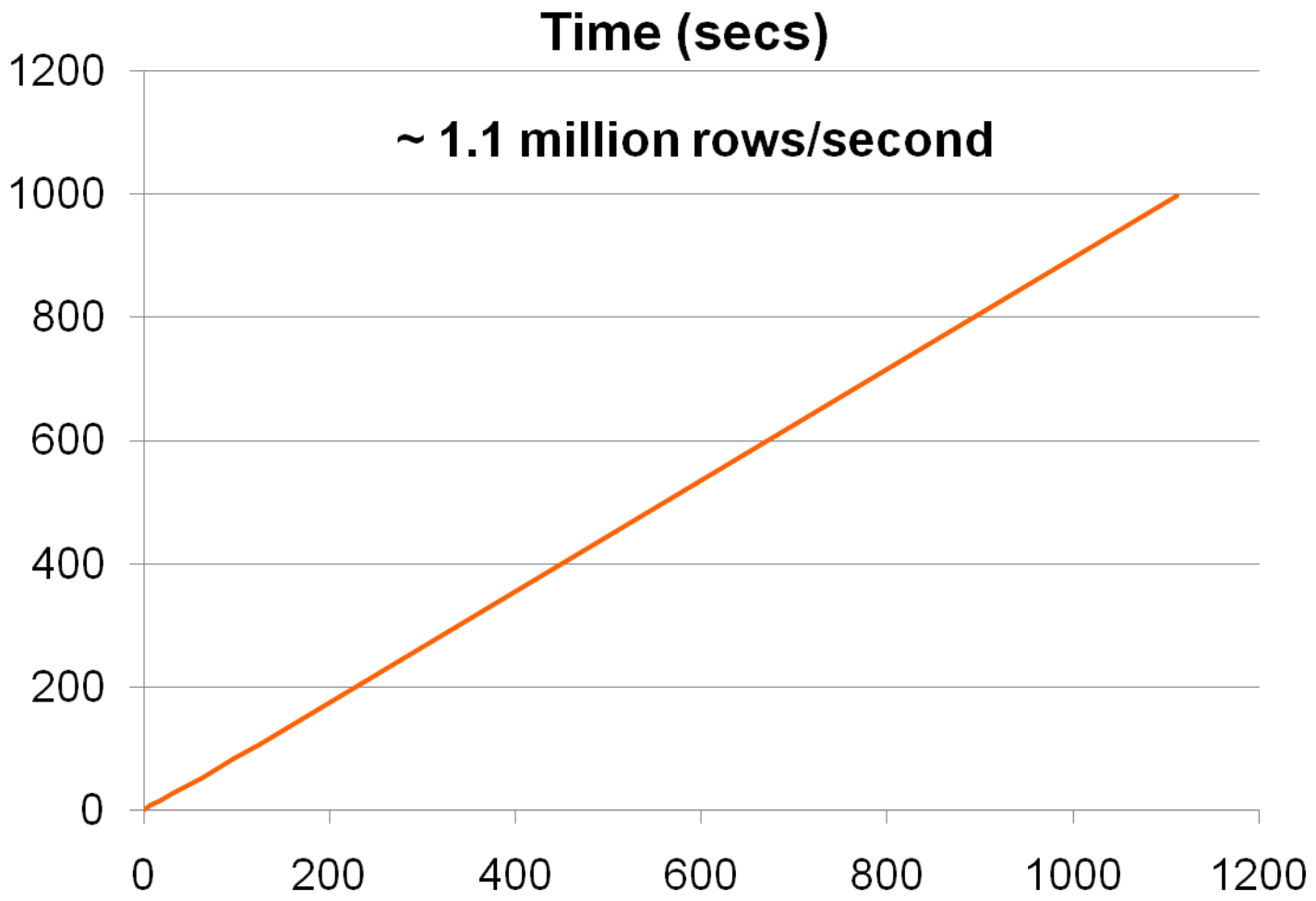
Distributed Import of Airline Data

- Copy the 22 CSV files to the 5 nodes, keeping rough balance in sizes
- Two pass import process:
  - Pass1: Import/clean/transform/append
  - Pass 2: Recode factors whose levels differ
- Import time: about **3 min 20 secs** on 5 node cluster (about 17 minutes on single node)

Scalability of RevoScaleR with Rows

Regression, 1 million - 1.1 billion rows, 443 betas

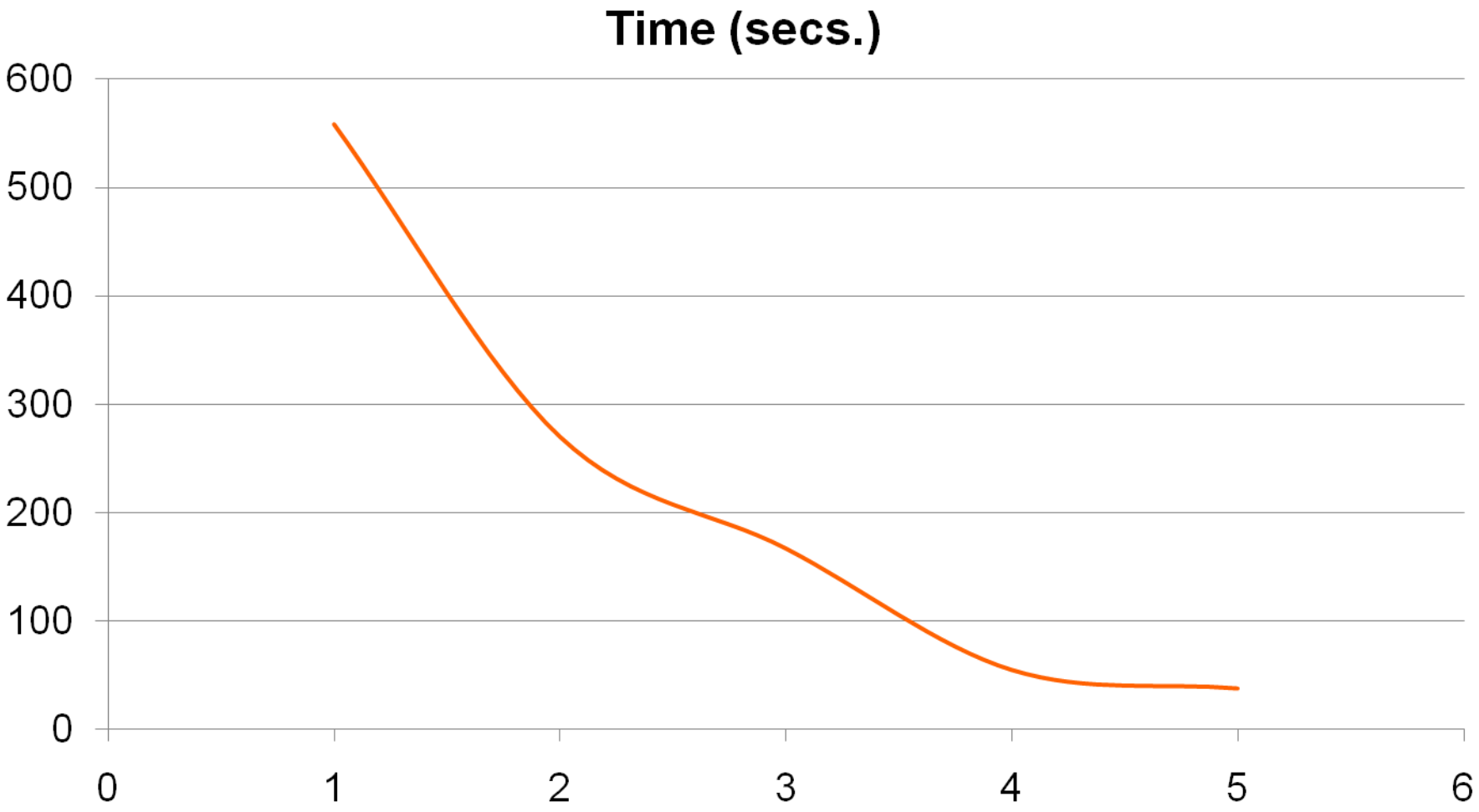
(4 core laptop)



Scalability of RevoScaleR with Nodes

Regression, 1 billion rows, 443 betas

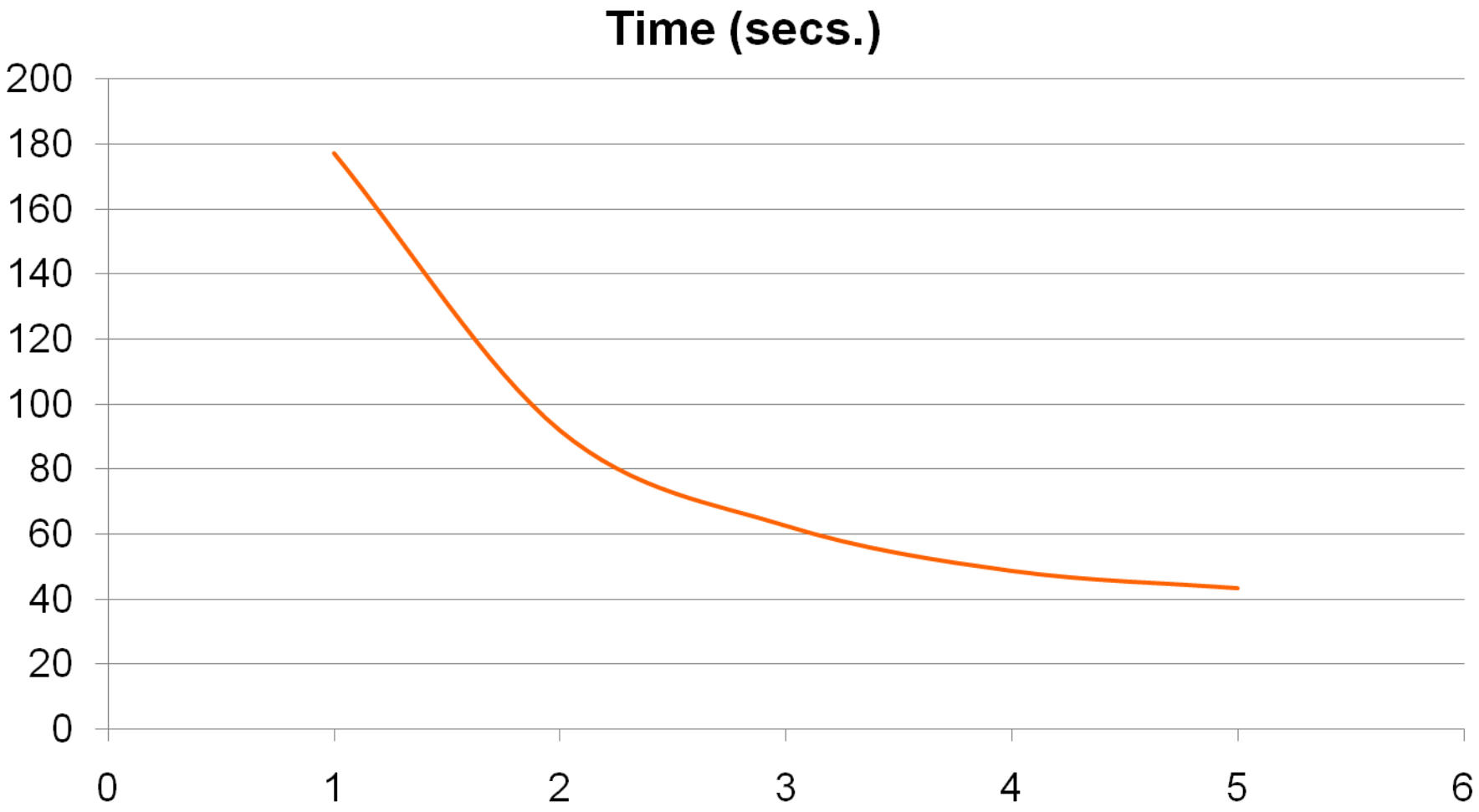
(1 to 5 nodes, 4 cores per node)



Scalability of RevoScaleR with Nodes

Logit, 123.5 million rows, 443 betas, 5 iterations

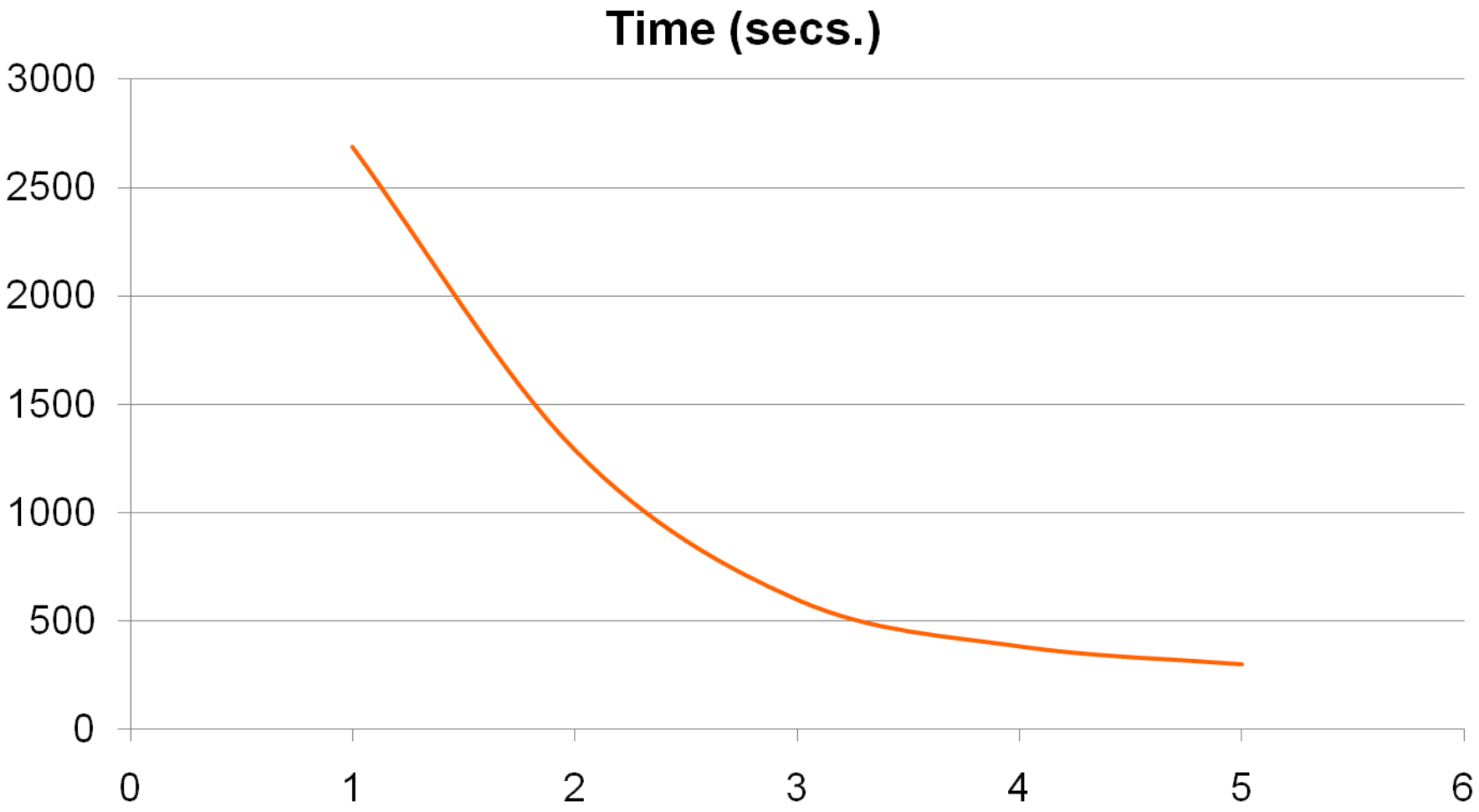
(1 to 5 nodes, 4 cores per node)



Scalability of RevoScaleR with Nodes

Logit, 1 billion rows, 443 betas, 5 iterations

(1 to 5 nodes, 4 cores per node)



Comparative benchmarks

- SAS HPA benchmarks:
  - Logit, billion rows, 32 nodes, 384 cores, in-memory, "just a few" parameters: 80 secs.
  - Regression, 50 million rows, 24 nodes, in-memory, 1,800 parameters: 42 secs.
- ScaleR: 1 billion rows, 5 nodes, 20 cores ($5K)
  - rxLogit, 7 parameters: 43.4 secs
  - rxLinMod, 7 parameters: 4.1 secs
  - rxLinMod, 1,848 params: 11.9 secs
  - rxLogit, 1,848 params: 111.5 secs
  - rxLinMod, 13,537 params: 89.8 secs.

Contact information

Lee Edlefsen

lee@revolutionanalytics.com